Homework

Visit course website, click on Resources and then on Code Style Guidelines. Study

4.2 Keep methods short
4.3 Use statement-comments ...
4.4 Use returns to simplify method structure
4.6 Declare local variables close to first use ...

Scope of local variable

```java
/** Return middle value of b, c, d (no ordering assumed) */
public static int middle(int b, int c, int d) {
    if (b > c) {
        int temp = b;
        b = c;
        c = temp;
    } // { b <= c }
    if (d <= b) {
        return b;
    } // { b < d and b <= c }
    return Math.min(c, d);
}
```

Scope of local variable (where it can be used): from its declaration to the end of the block in which it is declared.

Local variables

```java
/** Return middle value of b, c, d (no ordering assumed) */
public static int middle(int b, int c, int d) {
    if (b > c) {
        int temp = b;
        b = c;
        c = temp;
    } // { b <= c }
    if (d <= b) {
        return b;
    } // { b < d and b <= c }
    return Math.min(c, d);
}
```

Parameter: variable declared in () of method header

```
middle(8, 6, 7)
```

All parameters and local variables are created when a call is executed, before the method body is executed. They are destroyed when method body terminates.

Principle about placement of declaration

```java
/** Return middle value of b, c, d (no ordering assumed) */
public static int middle(int b, int c, int d) {
    int temp;
    if (b > c) {
        temp = b;
        b = c;
        c = temp;
    } // { b <= c }
    if (d <= b) {
        return b;
    } // { b < d and b <= c }
    return Math.min(c, d);
}
```

Principle: Declare a local variable as close to its first use as possible.
**Assertions promote understanding**

```java
/** Return middle value of b, c, d (no ordering assumed) */
public static int middle(int b, int c, int d) {
    if (b > c) {
        int temp = b;
        b = c;
        c = temp;
    } // { b <= c }
    if (d <= b) {
        return b;
    } // { b < d and b <= c }
    return Math.min(c, d);
}
```

**Bottom-up/overriding rule**

Which method `toString()` is called by `c.toString()`?

**Overriding rule or bottom-up rule:** To find out which is used, start at the bottom of the object and search upward until a matching one is found.

```java
toString() { … }
```

**Inside-out rule**

Inside-out rule: Code in a construct can reference any names declared in that construct, as well as names that appear in enclosing constructs. (If name is declared twice, the closer one prevails.)

```java
Person@0.n is this variable
```

**Parameters participate in inside-out rule**

Parameter `n` blocks reference to field `n`. (`n` is a "shadowed" variable)

```java
toString() { … }
```

**About super**

Within a subclass object, `super` refers to the partition above the one that contains `super`.

```java
toString() { … }
```
**Calling a constructor from a constructor**

```java
public class Time {
    private int hr; // hour of day, 0..23
    private int min; // minute of hour, 0..59

    /** Constructor: instance with h hours and m minutes */
    public Time(int h, int m) {
        hr = h;
        min = m;
    }

    /** Constructor: instance with m minutes */
    public Time(int m) {
        hr = m / 60;
        min = m % 60;
    }
}
```

---

**Initialize superclass fields first**

Class `Employee` contains info that is common to all employees — name, start date, salary, etc.

- `getCompensation` gives the salary
- Executives also get a bonus.
- `getCompensation` is overridden to take this into account
- Could have other subclasses for part-timers, temporary workers, consultants, etc., each with a different `getCompensation`

---

**Principle: initialize superclass fields first**

```java
/** Constructor: employee with name n, year hired d, salary s */
public Employee(String n, int d, double s) {
    name= n;
    start= d;
    salary= s;
}
```

---

**Calling a constructor from a constructor**

```java
public class Time {
    private int hr; // hour of day, 0..23
    private int min; // minute of hour, 0..59

    /** Constructor: instance with h hours and m minutes */
    public Time(int h, int m) {
        this(h, m);
    }

    /** Constructor: instance with m minutes */
    public Time(int m) {
        hr = m / 60;
        min = m % 60;
    }
}
```

---

**Without OO ...**

Without OO, you would write a long involved method:

```java
public double getCompensation(...) {
    if (worker is an executive) {
        ...
    } else if (worker is part time) {
        ...
    } else if (worker is temporary) {
        ...
    } else {
        ...
    }
}
```

**Principle: initialize superclass fields first**

```java
/** Constructor: employee with name n, year hired d, salary s */
public Employee(String n, int d, double s) {
    name= n;
    start= d;
    salary= s;
}
```

---

**Call constructor in superclass**

```java
/** Constructor: employee with name n, year hired d, salary s */
public Employee(String n, int d, double s) {
    name= n;
    start= d;
    salary= s;
}
```

---

**Principle: initialize superclass fields first**

```java
/** Constructor: executive with name n, year hired d, salary of $50,000, bonus b */
public Executive(String n, int d, double b) {
    name= n;
    start= d;
    salary= 50000;
    bonus= b;
}
```
Principle: initialize superclass fields first

```java
/** Constructor: employee with name n, year hired d, salary s */
public Employee(String n, int d, double s)

/** Constructor: executive with name n, year hired d, salary of $50,000, bonus b */
public Executive(String n, int d, double b) {
    super
    Employee(n, d, 50000);
    bonus= b;
}

To call a superclass constructor, use super(…)
```

```java
/** Constructor: executive with name n, year hired d, salary of $50,000, bonus b */
public Executive(String n, int d, double b) {
    super
    Employee(n, d, 50000);
    bonus= b;
}
```

To call a superclass constructor, use `super(…)`.

If you don’t put one in, Java silently inserts this one:
```
super();
```